



UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
10/772,992	02/05/2004	James S. Miller	13768.493	5389
47973 7590 11/23/2007 WORKMAN NYDEGGER/MICROSOFT 1000 EAGLE GATE TOWER 60 EAST SOUTH TEMPLE SALT LAKE CITY, UT 84111			EXAMINER WANG, BEN C	
			ART UNIT 2192	PAPER NUMBER
			MAIL DATE 11/23/2007	DELIVERY MODE PAPER

Please find below and/or attached an Office communication concerning this application or proceeding.

The time period for reply, if any, is set in the attached communication.

Office Action Summary

Application No.

10/772,992

Applicant(s)

MILLER ET AL.

Examiner

Ben C. Wang

Art Unit

2192

-- The MAILING DATE of this communication appears on the cover sheet with the correspondence address --

Period for Reply

A SHORTENED STATUTORY PERIOD FOR REPLY IS SET TO EXPIRE 3 MONTH(S) OR THIRTY (30) DAYS, WHICHEVER IS LONGER, FROM THE MAILING DATE OF THIS COMMUNICATION.

- Extensions of time may be available under the provisions of 37 CFR 1.136(a). In no event, however, may a reply be timely filed after SIX (6) MONTHS from the mailing date of this communication.
- If NO period for reply is specified above, the maximum statutory period will apply and will expire SIX (6) MONTHS from the mailing date of this communication.
- Failure to reply within the set or extended period for reply will, by statute, cause the application to become ABANDONED (35 U.S.C. § 133). Any reply received by the Office later than three months after the mailing date of this communication, even if timely filed, may reduce any earned patent term adjustment. See 37 CFR 1.704(b).

Status

- 1) ☒ Responsive to communication(s) filed on 9-12-2007.
- 2a) ☐ This action is **FINAL**. 2b) ☒ This action is non-final.
- 3) ☐ Since this application is in condition for allowance except for formal matters, prosecution as to the merits is closed in accordance with the practice under *Ex parte Quayle*, 1935 C.D. 11, 453 O.G. 213.

Disposition of Claims

- 4) ☒ Claim(s) 1-27 is/are pending in the application.
- 4a) Of the above claim(s) _____ is/are withdrawn from consideration.
- 5) ☐ Claim(s) _____ is/are allowed.
- 6) ☒ Claim(s) 1-27 is/are rejected.
- 7) ☐ Claim(s) _____ is/are objected to.
- 8) ☐ Claim(s) _____ are subject to restriction and/or election requirement.

Application Papers

- 9) ☐ The specification is objected to by the Examiner.
- 10) ☐ The drawing(s) filed on _____ is/are: a) ☐ accepted or b) ☐ objected to by the Examiner.
- Applicant may not request that any objection to the drawing(s) be held in abeyance. See 37 CFR 1.85(a).
- Replacement drawing sheet(s) including the correction is required if the drawing(s) is objected to. See 37 CFR 1.121(d).
- 11) ☐ The oath or declaration is objected to by the Examiner. Note the attached Office Action or form PTO-152.

Priority under 35 U.S.C. § 119

- 12) ☐ Acknowledgment is made of a claim for foreign priority under 35 U.S.C. § 119(a)-(d) or (f).
- a) ☐ All b) ☐ Some * c) ☐ None of:
1. ☐ Certified copies of the priority documents have been received.
2. ☐ Certified copies of the priority documents have been received in Application No. _____.
3. ☐ Copies of the certified copies of the priority documents have been received in this National Stage application from the International Bureau (PCT Rule 17.2(a)).
- * See the attached detailed Office action for a list of the certified copies not received.

Attachment(s)

- 1) ☒ Notice of References Cited (PTO-892)
- 2) ☐ Notice of Draftsperson's Patent Drawing Review (PTO-948)
- 3) ☒ Information Disclosure Statement(s) (PTO/SB/08)
Paper No(s)/Mail Date 10-31-2007.
- 4) ☐ Interview Summary (PTO-413)
Paper No(s)/Mail Date. _____.
- 5) ☐ Notice of Informal Patent Application
- 6) ☐ Other: _____.

DETAILED ACTION

1. A request for continued examination under 37 CFR 1.114, including the fee set forth in 37 CFR 1.17(e), was filed in this application after final rejection. Since this application is eligible for continued examination under 37 CFR 1.114, and the fee set forth in 37 CFR 1.17(e) has been timely paid, the finality of the previous Office action has been withdrawn pursuant to 37 CFR 1.114. Applicant's submission filed on September 12, 2007 has been entered.

2. Applicant's amendment dated September 12, 2007, responding to the July 12, 2007 Office action provided in the rejection of claims 1-27, wherein claims 1, 5-6, 8-11, 13-14, and 16-27 have been amended.

Claims 1-27 remain pending in the application and which have been fully considered by the examiner.

The Art Unit number and the date of response to the Office action are incorrect. The Art Unit number should be 2192 and the date should be July 12, 2007. Please correct them accordingly.

Applicant's arguments with respect to claims rejection have been fully considered but are moot in view of the new grounds of rejection – see *Eisenbach et al.*, art made of record, as applied hereto.

Art Unit: 2192

Claim Rejections – 35 USC § 102(a)

The following is a quotation of the appropriate paragraphs of 35 U.S.C. 102(a) that form the basis for the rejections under this section made in this office action:

A person shall be entitled to a patent unless –

(a) the invention was known or used by others in this country, or patented or described in a printed publication in this or a foreign country, before the invention thereof by the applicant for a patent.

3. Claims 22-24, and 27 are rejected under 35 U.S.C. 102(a) as being anticipated by Eisenbach et al. (*Managing the Evolution of .NET™ Programs*, 2003, IFIP International Federation for Information Processing 2003 (hereinafter 'Eisenbach' - art made of record)).

4. **As to claim 22** (Currently Amended), Eisenbach discloses in a computerized system that includes one or more program components including one or more requesting components that can request to access one or more target components in the computerized system, a method of automatically managing access of one or more versions of computer executable target component (e.g., Sec. 1 of Introduction, 3rd Par. – There is another way to solve these problems, and that is by allowing the system to keep multiple versions of the same DLL such that each application is linked to a compatible version) such that a computer-executable requesting component that accesses the target component continues to operate effectively after the target component has been upgraded with newer versions thereof, comprising the acts of:

- identifying that a requesting component is configured to execute a computer-executable target component (e.g., P. 186, 4th Par., Lines 7-9 – The Fusion utility in .NET is configured to find, according to some policy, the appropriate component and to pass its pathname to the Library Loader);
- automatically identifying a versioning policy in at least an available existing version of the target component and a versioning policy in an available, previously installed version of the target component (e.g., Sec. 5 of Conclusions, 1st Par. – but a large part lies in the versioning strategy of the linking mechanism developed by the run-time system; 2nd Par. – we have looked at this mechanism as deployed by current .NET development tools. We have shown that if a discipline is kept over which components can be added to the Global Assembly Cache, DLL Hell can be avoided; Consequently, the GAC is not usable as a cache for evolving components. As a consequence we have developed our prototype tool, to work alongside and within .NET and to demonstrate what could be attainable); and
- automatically determining that only one of the available versions of the target component should remain on the system based on any of the identified versioning policies corresponding to at least the existing version of the target component and the previously installed version of the target component (e.g., Abstract, 2nd Par. – We want to see whether the dynamic linking mechanism implemented in Microsoft's .NET™ environment could be utilized to maintain multiple versions of components. A formal model was developed to assist in

understanding the .NET™ mechanism and in describing our way of dealing with multiple versions. This showed that .NET™ incorporates all the features necessary to implement such a scheme and we constructed a tool to do so).

5. **As to claim 23** (Currently Amended) (incorporating the rejection in claim 22), Eisenbach discloses the method wherein the existing version of the target component includes a versioning value and a servicing value, the method further comprising receiving an updated servicing of the existing version of target component over a network from a network service provider (e.g., Fig. 4 – Distributed Dejavue.NET architecture, element of ‘Distributed Server’); and automatically overwriting the existing version of the target component, wherein the existing version of the target component reflects the versioning value and a new servicing value (e.g., P. 186, 4th Par. – The design of the .NET assembly reveals some details about how this is accomplished, with each assembly carrying versioning information structured into four fields – Major, Minor (e.g., servicing value), Revision and Build. This allows many versions of the same DLL to co-exist in the system).

6. **As to claim 24** (Currently Amended) (incorporating the rejection in claim 22), Eisenbach discloses the method further comprising, upon reviewing the versioning policy of one or more new versions of the target component, automatically adding the new versions of the target component to the system without removing any of the previously installed versions of the target component (e.g., Sec. 1 of Introduction, 3rd

Art Unit: 2192

Par. – There is another way to solve these problems, and that is by allowing the system to keep multiple versions of the same DLL such that each application is linked to a compatible version).

7. **As to claim 27** (Currently Amended), Eisenbach discloses in a computerized system including one or more requesting components that are configured to access one or more target components in the computerized system, a computer program storage product having computer-executable instructions stored thereon that, when executed, cause one or more processors in the computerized system to execute a method of automatically managing access of one or more versions of computer-executable target component (e.g., Sec. 1 of Introduction, 3rd Par. – There is another way to solve these problems, and that is by allowing the system to keep multiple versions of the same DLL such that each application is linked to a compatible version) such that a computer-executable requesting component that accesses the computer-executable target component continues to operate effectively after the target component has been upgraded with newer versions thereof, comprising the acts of:

- identifying that a requesting component is configured to execute a computer-executable target component (e.g., P. 186, 4th Par., Lines 7-9 – The Fusion utility in .NET is configured to find, according to some policy, the appropriate component and to pass its pathname to the Library Loader);
- automatically identifying a versioning policy in at least an available existing version of the target component and a versioning policy in an available, previously installed

Art Unit: 2192

version of the target component (e.g., Sec. 5 of Conclusions, 1st Par. – but a large part lies in the versioning strategy of the linking mechanism developed by the run-time system; 2nd Par. – we have looked at this mechanism as deployed by current .NET development tools. We have shown that if a discipline is kept over which components can be added to the Global Assembly Cache, DLL Hell can be avoided; Consequently, the GAC is not usable as a cache for evolving components. As a consequence we have developed our prototype tool, to work alongside and within .NET and to demonstrate what could be attainable); and

- automatically determining that only one of the available versions of the target component should remain on the system based on any of the identified versioning policies corresponding to at least the existing version of the target component and the previously installed version of the target component (e.g., Abstract, 2nd Par. – We want to see whether the dynamic linking mechanism implemented in Microsoft's .NET™ environment could be utilized to maintain multiple versions of components. A formal model was developed to assist in understanding the .NET™ mechanism and in describing our way of dealing with multiple versions. This showed that .NET™ incorporates all the features necessary to implement such a scheme and we constructed a tool to do so).

Art Unit: 2192

Claim Rejections – 35 USC § 103(a)

The following is a quotation of 35 U.S.C. 103(a) which forms the basis for all obviousness rejections set forth in this office action:

(a) A patent may not be obtained though the invention is not identically disclosed or described as set forth in section 102 of this title, if the differences between the subject matter sought to be patented and the prior art are such that the subject matter as a whole would have been obvious at the time the invention was made to a person having ordinary skill in the art to which said subject matter pertains. Patentability shall not be negated by the manner in which the invention was made.

8. Claims 1-6, 9, 12-15, 20-21, and 25-26 are rejected under 35 U.S.C. 103(a) as being unpatentable over Eisenbach in view of Mike Gunderloy (*Managing Versions of an Application*, Feb. 2002, Lark Group, Inc., pp. 1-6) (hereinafter 'Gunderloy')

9. **As to claim 1** (Currently Amended), Eisenbach discloses in a computerized system that includes one or more program computer-executable program components including one or more computer-executable requesting components configured to execute one or more computer-executable target components in the computerized system, a method of automatically providing a computer-executable requesting component with access to an automatically determined version of a computer-executable target component upon request, comprising the acts of:

- receiving a request from a requesting component for access by the requesting component of a computer-executable target component, wherein the request includes an indication of the lowest possible version of the target component that the requesting component can accept (e.g., Sec. 1 of Introduction, 2nd Par. – This dynamic evolution is a benefit provided that two conditions are met. Firstly,

Art Unit: 2192

successive versions of the DLL must maintain backward compatibility (*lowest possible version of the target component*). Secondly, on any particular system, any given version of a DLL can only be replaced by a later version (*as high as the lowest possible version for backward compatibility*). Failure to meet the first condition results in the *upgrade* problem whilst failure to meet the second results in the *downgrade* problem. Many Microsoft™ support personnel report DLL hell as the single most significant user problem that they are called upon to respond to);

- upon receiving the request from the requesting component; identifying a versioning policy of the specified lowest possible version of the requested target component (e.g., Sec. 5 of Conclusions, 1st Par – but at larger part lies in the versioning strategy of the link mechanism deployed by the run-time system; 2nd Par. – we have looked at this mechanism as deployed by current .NET development tools. We have shown that if a discipline is kept over which components can be added to the Global Assembly Cache, DLL Hell can be avoided);
- identifying an appropriate version of the target component based on the versioning policy of the specified lowest possible version of the target component, wherein the appropriate version of the target component is at least as high as the lowest possible version (e.g., Sec. 1 of Introduction, 2nd Par. – This dynamic evolution is a benefit provided that two conditions are met. Firstly, successive versions of the DLL must maintain backward compatibility (*lowest possible version of the target component*). Secondly, on any particular system, any given version of a DLL can only be replaced by a later version (*as high as the lowest possible version*)).

Art Unit: 2192

Eisenbach does not explicitly disclose providing the requesting component with access to the appropriate version of the target component, wherein the requesting component executes the identified and provided target component.

However, in an analogous art of *Managing Versions of an Application*, Gunderloy discloses providing the requesting component with access to the appropriate version of the target component, wherein the requesting component executes the identified and provided target component (e.g., P. 3, Sec. of "Practice Modifying Version Information at Runtime", 1st Par. – you'll see that even though there is a more recent version of the library installed, the older version will be used by the client application until you explicitly construct an application configuration file; P. 3, Sec. of "Finding the Correct Version at Runtime" – when you run a Microsoft .NET™ application that uses an external component, the Microsoft .NET™ runtime checks four places to determine which version of the component to load: 1) the original version from the assembly manifest is the default version to load, 2) the runtime then checks for the presence of an application policy file that overrides the version information for this application only, 3) the runtime then checks for the presence of a publisher policy file that overrides the version information for this component in all applications, 4) the runtime then checks for the presence of an administrator policy file that overrides the version information for this component system-wide; P. 3, Sec. of "Policy Files" – there are three policy files that the runtime checks for version information 1) the application policy files has the same name as the application plus the extension ".config" and resides in the same directory as the application, 2) the publisher policy file is distributed by a component publisher together

Art Unit: 2192

with a new version of a component, 3) the machine configuration file is named *Machine.config* and is stored in the Config. directory beneath the directory where the Microsoft .NET™ runtime is installed; P. 5, Sec. of "Summary", - Microsoft .NET™ allows very fine control of version information. By default, an assembly will use only the versions of components that were available when the assembly was compiled. But as the developer of an application, the publisher of a component, or the administrator of a system you can override this policy and specify the versions to be used on a component-by-component basis).

Therefore, it would have been obvious to one of ordinary skill in the art, at the time the invention was made to combine the teachings of Gunderloy into the Eisenbach's system to further provide the requesting component with access to the appropriate version of the target component, wherein the requesting component executes the identified and provided target component in Eisenbach system.

The motivation is that it would further enhance the Eisenbach's system by taking, advancing and/or incorporating Gunderloy's system which offers significant advantages that Microsoft .NET™ allows very fine control of version information. By default, an assembly will use only the versions of components that were available when the assembly was compiled. But as the developer of application, the publisher of a component, or the administrator of a system you can override this policy and specify the versions to be used on a component-by-component basis as once suggested by Gunderloy (e.g., Sec. Summary).

Art Unit: 2192

10. **As to claim 2** (Original) (incorporating the rejection in claim 1), Gunderloy discloses the method wherein the requested version of the target component is one of a library component and a platform component (e.g., P. 3, Sec. of "Practice Modifying Version Information at Runtime", 1st Par. through P. 5, Sec. of "Try It Out"; P. 3, Sec. of "Policy Files").

11. **As to claim 3** (Original) (incorporating the rejection in claim 1), Gunderloy discloses the method wherein identifying an appropriate version of the target component comprises identifying a more recent version of the target component in response to a request for an earlier version of the target even though the more recent version and the earlier version are both accessible to the computerized system (e.g., P. 2, Sec. of "Version Compatibility", 1st Par. – the main use of version information is to help applications and developers determine whether a particular version of a component is compatible with an application that makes use of the component; by default, Microsoft .NET™ uses only the versions of components that were present when an assembly was compiled; 2nd Par. – for example, if a Visual Basic™ 6.0 application calls a component named *xyz.dll*, and version 1 was present on the computer at the time that the application was compiled, and later version 1.1 of *xyz.dll* is installed, the application will automatically call version 1.1).

12. **As to claim 4** (Original) (incorporating the rejection in claim 2), Gunderloy discloses the method identifying a more recent version of the target component in

Art Unit: 2192

response to a request for an earlier version of the target even though the more recent version and the earlier version are both accessible to the computerized system comprises identifying a more recent version of a platform component even though an earlier version of the platform component remained on the system when the more recent version was received at the computerized system (e.g., P. 2, Sec. of "Version Compatibility", 1st Par. – the main use of version information is to help applications and developers determine whether a particular version of a component is compatible with an application that makes use of the component; by default, Microsoft .NET™ uses only the versions of components that were present when an assembly was compiled; 2nd Par. – for example, if a Visual Basic™ 6.0 application calls a component named *xyz.dll*, and version 1 was present on the computer at the time that the application was compiled, and later version 1.1 of *xyz.dll* is installed, the application will automatically call version 1.1).

13. **As to claim 5** (Currently Amended) (incorporating the rejection in claim 1), Eisenbach discloses the method wherein further comprising an act of identifying the versioning policy of the specified lowest possible version of the target component when the specified lowest possible version of the target component is added to the computerized system (e.g., e.g., Sec. 1 of Introduction, 2nd Par. – This dynamic evolution is a benefit provided that two conditions are met. Firstly, successive versions of the DLL must maintain backward compatibility (lowest possible version of the target component). Secondly, on any particular system, any given version of a DLL can only

Art Unit: 2192

be replaced by a later version (as high as the lowest possible version for backward compatibility). Failure to meet the first condition results in the *upgrade* problem whilst failure to meet the second results in the *downgrade* problem. Many Microsoft™ support personnel report DLL hell as the single most significant user problem that they are called upon to respond to).

14. **As to claim 6** (Currently Amended) (incorporating the rejection in claim 1), Eisenbach discloses the method wherein further comprising an act of storing, in the requesting component version information that identifies the specified lowest possible version of the target component in the requesting component when the requesting component is one or more of compiled, configured, installed, and run on the computerized system (e.g., Sec. 1 of Introduction, 2nd Par. – This dynamic evolution is a benefit provided that two conditions are met. Firstly, successive versions of the DLL must maintain backward compatibility (lowest possible version of the target component). Secondly, on any particular system, any given version of a DLL can only be replaced by a later version (as high as the lowest possible version for backward compatibility). Failure to meet the first condition results in the *upgrade* problem whilst failure to meet the second results in the *downgrade* problem. Many Microsoft™ support personnel report DLL hell as the single most significant user problem that they are called upon to respond to).

Art Unit: 2192

15. **As to claim 9** (Currently Amended) (incorporating the rejection in claim 8), Gunderloy discloses the method wherein the automatically determined appropriate version of the target component is different from the requested specific version of the target component that was requested (e.g., P. 2, Sec. of "Version Compatibility", 1st Par. – the main use of version information is to help applications and developers determine whether a particular version of a component is compatible with an application that makes use of the component; by default, Microsoft .NET™ uses only the versions of components that were present when an assembly was compiled; 2nd Par. – for example, if a Visual Basic™ 6.0 application calls a component named *xyz.dll*, and version 1 was present on the computer at the time that the application was compiled, and later version 1.1 of *xyz.dll* is installed, the application will automatically call version 1.1).

16. **As to claim 12** (Original) (incorporating the rejection in claim 1), Gunderloy discloses the method wherein the versioning policy is inserted into computer-executable instructions in the target component prior to one of installing, configuring, and executing the target component on the computerized system (e.g., P. 3, Sec. of "Policy Files" – policy files are XML files that specify the version of a component to use; the general structure of a policy file follows this template - <configuration> .. </configuration>; P. 2, Sec. of "Version Compatibility", 1st Par. – by default, Microsoft.NET uses only the versions of components that were present when an assembly was compiled; 3rd Par. – by tying assemblies to the versions of components that were present at compile time,

Art Unit: 2192

the .NET Framework™ helps to avoid the “DLL Hell” syndrome in which new versions of shared libraries break existing application).

17. **As to claim 13** (Currently Amended) (incorporating the rejection in claim 1), Gunderloy discloses the method wherein the versioning policy is further identified in a plurality of version of the target component on the computerized system (e.g., P. 3, Sec. of “Policy Files” – the version for the *oldVersion* attribute can either be a single version number such as “1.0.0.0”).

18. **As to claim 14** (Currently Amended) (incorporating the rejection in claim 12), Gunderloy discloses the method wherein each versioning policy in each version of the target component identifies a specific version of the requesting component configured to access that target component (e.g., P. 3, Sec. of “Policy Files” – the version for the *oldVersion* attribute can either be a single version number such as “1.0.0.0” or a range of version numbers such as “1.0.0.0-3.0.0.0”).

19. **As to claim 15** (Original) (incorporating the rejection in claim 1), Gunderloy discloses the method further comprising identifying a component scope that is associated with the target component (e.g., P. 3, Sec. of “Policy Files” – the version for the *oldVersion* attribute can either be a single version number such as “1.0.0.0” or a range of version numbers such as “1.0.0.0-3.0.0.0”).

Art Unit: 2192

20. **As to claim 20** (Currently Amended), Eisenbach discloses in a computerized system that includes one or more computer-executable program components including one or more computer-executable requesting components that can request to access one or more computer-executable target components in the computerized system, a method of automatically providing a computer-executable requesting component with access to an automatically determined version of a target component, comprising:

- receiving a request from a requesting component for access by the requesting component of a target component, wherein the request includes an indication of the lowest possible version of the target component that the requesting component can accept (e.g., Sec. 1 of Introduction, 2nd Par. – This dynamic evolution is a benefit provided that two conditions are met. Firstly, successive versions of the DLL must maintain backward compatibility (*lowest possible version of the target component*). Secondly, on any particular system, any given version of a DLL can only be replaced by a later version (*as high as the lowest possible version for backward compatibility*). Failure to meet the first condition results in the *upgrade* problem whilst failure to meet the second results in the *downgrade* problem. Many Microsoft™ support personnel report DLL hell as the single most significant user problem that they are called upon to respond to).

Eisenbach does not explicitly disclose a step for, upon receiving the request from the requesting component, determining an appropriate version of the requested target component based on a versioning policy corresponding to the requested target component; and allowing access to an appropriate version of the requested target

Art Unit: 2192

component such that the requesting component accesses the appropriate target component as it has been configured to do so, and such that the requesting component does not fail when requesting access to a component that has been upgraded.

However, in an analogous art of *Managing Versions of an Application*, Gunderloy discloses a step for, upon receiving the request from the requesting component, determining an appropriate version of the requested target component based on a versioning policy corresponding to the requested target component (e.g., P. 3, Sec. of “Practice Modifying Version Information at Runtime”, 1st Par. – you’ll see that even though there is a more recent version of the library installed, the older version will be used by the client application until you explicitly construct an application configuration file; P. 3, Sec. of “Finding the Correct Version at Runtime” – when you run a Microsoft .NET™ application that uses an external component, the Microsoft .NET™ runtime checks four places to determine which version of the component to load: 1) the original version from the assembly manifest is the default version to load, 2) the runtime then checks for the presence of an application policy file that overrides the version information for this application only, 3) the runtime then checks for the presence of a publisher policy file that overrides the version information for this component in all applications, 4) the runtime then checks for the presence of an administrator policy file that overrides the version information for this component system-wide; P. 3, Sec. of “Policy Files” – there are three policy files that the runtime checks for version information 1) the application policy files has the same name as the application plus the extension “.config” and resides in the same directory as the application, 2) the publisher

Art Unit: 2192

policy file is distributed by a component publisher together with a new version of a component, 3) the machine configuration file is named *Machine.config* and is stored in the Config. directory beneath the directory where the Microsoft .NET™ runtime is installed; P. 5, Sec. of "Summary", - Microsoft .NET™ allows very fine control of version information. By default, an assembly will use only the versions of components that were available when the assembly was compiled. But as the developer of an application, the publisher of a component, or the administrator of a system you can override this policy and specify the versions to be used on a component-by-component basis), and allowing access to an appropriate version of the requested target component such that the requesting component accesses the appropriate target component as it has been configured to do so, and such that the requesting component does not fail when requesting access to a component that has been upgraded (e.g., P. 2, Sec. of "Version Compatibility", 1st Par. – the main use of version information is to help applications and developers determine whether a particular version of a component is compatible with an application that makes use of the component; by default, Microsoft .NET™ uses only the versions of components that were present when an assembly was compiled; 2nd Par. – for example, if a Visual Basic™ 6.0 application calls a component named *xyz.dll*, and version 1 was present on the computer at the time that the application was compiled, and later version 1.1 of *xyz.dll* is installed, the application will automatically call version 1.1).

Therefore, it would have been obvious to one of ordinary skill in the art, at the time the invention was made to combine the teachings of Gunderloy into the Eisenbach's

Art Unit: 2192

system to further provide a step for, upon receiving the request from the requesting component, determining an appropriate version of the requested target component based on a versioning policy corresponding to the requested target component; and allowing access to an appropriate version of the requested target component such that the requesting component accesses the appropriate target component as it has been configured to do so, and such that the requesting component does not fail when requesting access to a component that has been upgraded in Eisenbach system.

The motivation is that it would further enhance the Eisenbach's system by taking, advancing and/or incorporating Gunderloy's system which offers significant advantages that Microsoft .NET™ allows very fine control of version information. By default, an assembly will use only the versions of components that were available when the assembly was compiled. But as the developer of application, the publisher of a component, or the administrator of a system you can override this policy and specify the versions to be used on a component-by-component basis as once suggested by Gunderloy (e.g., Sec. Summary).

21. **As to claim 21** (Currently Amended) (incorporating the rejection in claim 20), Gunderloy discloses the method wherein the step for allowing access to an appropriate version of the requested target component comprises the corresponding acts of: upon receiving the request from the requesting component, identifying a versioning policy of the specified version of the requested target component; identifying an appropriate version of the target component based on the versioning policy of the specified target

Art Unit: 2192

component; providing the requesting component with access to the appropriate version of the target component, wherein the requesting component executes the identified and provided target component (e.g., P. 3, Sec. of "Practice Modifying Version Information at Runtime", 1st Par. – you'll see that even though there is a more recent version of the library installed, the older version will be used by the client application until you explicitly construct an application configuration file; P. 3, Sec. of "Finding the Correct Version at Runtime" – when you run a Microsoft .NET™ application that uses an external component, the Microsoft .NET™ runtime checks four places to determine which version of the component to load: 1) the original version from the assembly manifest is the default version to load, 2) the runtime then checks for the presence of an application policy file that overrides the version information for this application only, 3) the runtime then checks for the presence of a publisher policy file that overrides the version information for this component in all applications, 4) the runtime then checks for the presence of an administrator policy file that overrides the version information for this component system-wide; P. 3, Sec. of "Policy Files" – there are three policy files that the runtime checks for version information 1) the application policy files has the same name as the application plus the extension ".config" and resides in the same directory as the application, 2) the publisher policy file is distributed by a component publisher together with a new version of a component, 3) the machine configuration file is named *Machine.config* and is stored in the Config. directory beneath the directory where the Microsoft .NET™ runtime is installed; P. 5, Sec. of "Summary", - Microsoft .NET™ allows very fine control of version information. By default, an assembly will use only the

Art Unit: 2192

versions of components that were available when the assembly was compiled. But as the developer of an application, the publisher of a component, or the administrator of a system you can override this policy and specify the versions to be used on a component-by-component basis).

22. **As to claim 25** (Currently Amended) (incorporating the rejection in claim 24), Eisenbach does not explicitly disclose the method wherein the automatic determination is based on whether the target component is one of a library component or a platform component.

However, in an analogous art of *Managing Versions of an Application*, Gunderloy discloses the method wherein the automatic determination is based on whether the target component is one of a library component or a platform component (e.g., P. 3, Sec. of "Policy Files" – the version for the *oldVersion* attribute can either be a single version number such as "1.0.0.0" or a range of version numbers such as "1.0.0.0-3.0.0.0").

Therefore, it would have been obvious to one of ordinary skill in the art, at the time the invention was made to combine the teachings of Gunderloy into the Eisenbach's system to further provide the method wherein the automatic determination is based on whether the target component is one of a library component or a platform component in Eisenbach system.

The motivation is that it would further enhance the Eisenbach's system by taking, advancing and/or incorporating Gunderloy's system which offers significant advantages

Art Unit: 2192

that Microsoft .NET™ allows very fine control of version information. By default, an assembly will use only the versions of components that were available when the assembly was compiled. But as the developer of application, the publisher of a component, or the administrator of a system you can override this policy and specify the versions to be used on a component-by-component basis as once suggested by Gunderloy (e.g., Sec. Summary).

23. **As to claim 26** (Currently Amended), Eisenbach discloses in a computerized system including one or more requesting components that are configured to access one or more target components in the computerized system, a computer storage product having computer-executable instructions stored thereon that, when executed, cause one or more processors in the computerized system to execute a method of automatically determined version of a computer-executable requesting component upon request, comprising the acts of:

- receiving a request from a requesting component for access by the requesting component of a computer-executable target component, wherein the request includes an indication of the lowest possible version of the target component that the requesting component can accept (e.g., Sec. 1 of Introduction, 2nd Par. – This dynamic evolution is a benefit provided that two conditions are met. Firstly, successive versions of the DLL must maintain backward compatibility (*lowest possible version of the target component*). Secondly, on any particular system, any given version of a DLL can only be replaced by a later version (*as high as the lowest*

possible version for backward compatibility). Failure to meet the first condition results in the *upgrade* problem whilst failure to meet the second results in the *downgrade* problem. Many Microsoft™ support personnel report DLL hell as the single most significant user problem that they are called upon to respond to);

- upon receiving the request from the requesting component (e.g., Sec. 5 of Conclusions, 1st Par – but at larger part lies in the versioning strategy of the link mechanism deployed by the run-time system; 2nd Par. – we have looked at this mechanism as deployed by current .NET development tools. We have shown that if a discipline is kept over which components can be added to the Global Assembly Cache, DLL Hell can be avoided), identifying a versioning policy of the specified lowest possible version of the requested target component;
- identifying an appropriate version of the target component based on the versioning policy of the specified lowest possible version of the target component, wherein the appropriate version of the target component is at least as high as the lowest possible version (e.g., Sec. 1 of Introduction, 2nd Par. – This dynamic evolution is a benefit provided that two conditions are met. Firstly, successive versions of the DLL must maintain backward compatibility (*lowest possible version of the target component*). Secondly, on any particular system, any given version of a DLL can only be replaced by a later version (*as high as the lowest possible version*)).

Eisenbach does not explicitly disclose providing the requesting component with access to the appropriate version of the target component, wherein the requesting component executes the identified and provided target component.

Art Unit: 2192

However, in an analogous art of *Managing Versions of an Application*, Gunderloy discloses providing the requesting component with access to the appropriate version of the target component, wherein the requesting component executes the identified and provided target component (e.g., P. 3, Sec. of "Practice Modifying Version Information at Runtime", 1st Par. – you'll see that even though there is a more recent version of the library installed, the older version will be used by the client application until you explicitly construct an application configuration file; P. 3, Sec. of "Finding the Correct Version at Runtime" – when you run a Microsoft .NET™ application that uses an external component, the Microsoft .NET™ runtime checks four places to determine which version of the component to load: 1) the original version from the assembly manifest is the default version to load, 2) the runtime then checks for the presence of an application policy file that overrides the version information for this application only, 3) the runtime then checks for the presence of a publisher policy file that overrides the version information for this component in all applications, 4) the runtime then checks for the presence of an administrator policy file that overrides the version information for this component system-wide; P. 3, Sec. of "Policy Files" – there are three policy files that the runtime checks for version information 1) the application policy files has the same name as the application plus the extension ".config" and resides in the same directory as the application, 2) the publisher policy file is distributed by a component publisher together with a new version of a component, 3) the machine configuration file is named *Machine.config* and is stored in the Config. directory beneath the directory where the Microsoft .NET™ runtime is installed; P. 5, Sec. of "Summary", - Microsoft .NET™

Art Unit: 2192

allows very fine control of version information. By default, an assembly will use only the versions of components that were available when the assembly was compiled. But as the developer of an application, the publisher of a component, or the administrator of a system you can override this policy and specify the versions to be used on a component-by-component basis).

Therefore, it would have been obvious to one of ordinary skill in the art, at the time the invention was made to combine the teachings of Gunderloy into the Eisenbach's system to further provide providing the requesting component with access to the appropriate version of the target component, wherein the requesting component executes the identified and provided target component in Eisenbach system.

The motivation is that it would further enhance the Eisenbach's system by taking, advancing and/or incorporating Gunderloy's system which offers significant advantages that Microsoft .NET allows very fine control of version information. By default, an assembly will use only the versions of components that were available when the assembly was compiled. But as the developer of application, the publisher of a component, or the administrator of a system you can override this policy and specify the versions to be used on a component-by-component basis as once suggested by Gunderloy (e.g., Sec. Summary).

Art Unit: 2192

24. Claims 7-8, 10-11, and 16-19 are rejected under 35 U.S.C. 103(a) as being unpatentable over Eisenbach in view of Gunderloy, and in view of Steven Pratschner (*Simplifying Deployment and Solving DLL Hell with the .NET Framework™*, Nov. 2001, *Microsoft Corporation™*, pp. 1-12) (hereinafter 'Pratschner')

25. **As to claim 7** (Previously Presented) (incorporating the rejection in claim 1), Eisenbach and Gunderloy do not explicitly disclose the method further comprising: identifying one or more requesting components that are able to access a prior version of the target component; identifying that none of the one or more requesting components are configured to request the prior version of the target component; and deleting the prior version of the target component.

However, in an analogous art of *Simplifying Deployment and Solving DLL Hell with the .NET Framework™*, Pratschner discloses are able to access a prior version of the target component; identifying that none of the one or more requesting components are configured to request the prior version of the target component; and deleting the prior version of the target component (e.g., P. 1, Sec. of "Introduction", 3rd Par. - .NET™ uses assemblies to solve versioning and deployment problems; in particular, how assemblies are structured, how they are named, and how compilers and the Common Language Runtime™ (CLR) use assemblies to record and enforce version dependencies between pieces of applications, and how applications and administrators can customize versioning behavior through what we call *version policies*; P. 7, Sec. of "Version Policy" P. 8, Sec. of "Default Version Policy", Sec. of "Custom Version Policy",

Art Unit: 2192

and Sec. of "Version Policy Levels" – including "Application –specific Policy", "Publisher Policy", and "Machine-wide Policy"; P. 9, Sec. of "Bypassing Publisher Policy").

Therefore, it would have been obvious to one of ordinary skill in the art, at the time the invention was made to combine the teachings of Pratschner into the Eisenbach-Gunderloy's system to further provide the method identifying one or more requesting components that are able to access a prior version of the target component; identifying that none of the one or more requesting components are configured to request the prior version of the target component; and deleting the prior version of the target component in Eisenbach-Gunderloy system.

The motivation is that it would further enhance the Eisenbach-Gunderloy's system by taking, advancing and/or incorporating Pratschner's system which offers significant advantages that the CLR records version information between pieces of an application and uses that information at run time to ensure that the proper version of a dependency is loaded; version policies can be used by both application developers and administrators to provide some flexibility in choosing which version of a given assembly is loaded as once suggested by Pratschner (e.g., P. 12, Sec. of "Summary").

26. **As to claim 8** (Currently Amended) (incorporating the rejection in claim 1), Eisenbach and Gunderloy do not explicitly disclose the method wherein the request further includes a request for a specific version of the target component, wherein the requested specific version is different from the lowest possible version of the target component.

However, in an analogous art of *Simplifying Deployment and Solving DLL Hell with the .NET Framework*[™], Pratschner discloses the method wherein the appropriate version of the target component is the version of the target component that was requested (e.g., P. 8, Sec. of “Default Version Policy” – when resolving a reference, the CLR takes the version from the calling assembly’s manifest and loads the version of the dependency with the exact same version number; in this way, the caller gets the exact version that he was built and tested against).

Therefore, it would have been obvious to one of ordinary skill in the art, at the time the invention was made to combine the teachings of Pratschner into the Eisenbach-Gunderloy’s system to further provide the method wherein the appropriate version of the target component is the version of the target component that was requested in Gunderloy system.

The motivation is that it would further enhance the Eisenbach-Gunderloy’s system by taking, advancing and/or incorporating Pratschner’s system which offers significant advantages that the CLR records version information between pieces of an application and uses that information at run time to ensure that the proper version of a dependency is loaded; version policies can be used by both application developers and administrators to provide some flexibility in choosing which version of a given assembly is loaded as once suggested by Pratschner (e.g., P. 12, Sec. of “Summary”).

27. **As to claim 10** (Currently Amended) (incorporating the rejection in claim 1), Eisenbach disclose further comprising receiving a plurality of new versions of the target

Art Unit: 2192

component, wherein each of the new versions of the target component are associated with a different versioning policy (e.g., Sec. 1 of Introduction, 3rd Par. – There is another way to solve these problems, and that is by allowing the system to keep multiple versions of the same DLL such that each application is linked to a compatible version).

28. **As to claim 11** (Currently Amended) (incorporating the rejection in claim 9), Pratschner discloses the method further comprising determining the appropriate version of the target component from among the specified lowest possible version of the target component and each of the plurality of new versions of the target component when the plurality of new versions of the target component (e.g., P. 1, Sec. of “Introduction”, 3rd Par. – the Common Language Runtime™ (CLR) uses assemblies to record and enforce version dependencies between pieces of an application; P. 5, Sec. of “Application-Private Assemblies, 2nd Par. – 3rd Par. – the CLR™ finds these assemblies through a process called probing; probing is simply a mapping of the assembly name to the name of the file that contains the manifest; specifically, the CLR™ takes the name of the assembly recorded in the assembly reference, appends “.dll” and looks for that file in the application directory; P. 8, Sec. of “Default Version Policy” – when resolving a reference, the CLR™ takes the version from the calling assembly’s manifest and loads the version of the dependency with the exact same version number; the first thing the CLR™ does when binding to a strongly named assembly is determine which version of the assembly to bind to).

Art Unit: 2192

29. **As to claim 16** (Currently Amended) (incorporating the rejection in claim 1), Eisenbach discloses the method wherein appropriate version of the target component is further automatically determined based on one of the identified component scope associated with the target component in addition to a determination of the lowest possible version that can be accepted (e.g., Sec. 1 of Introduction, 2nd Par. – This dynamic evolution is a benefit provided that two conditions are met. Firstly, successive versions of the DLL must maintain backward compatibility (*lowest possible version of the target component*). Secondly, on any particular system, any given version of a DLL can only be replaced by a later version (*as high as the lowest possible version for backward compatibility*). Failure to meet the first condition results in the *upgrade* problem whilst failure to meet the second results in the *downgrade* problem. Many Microsoft™ support personnel report DLL hell as the single most significant user problem that they are called upon to respond to).

30. **As to claim 17** (Currently Amended) (incorporating the rejection in claim 15), Pratschner discloses the method wherein the identified component scope specifies that access to the specified version of the target component is provided differently from the lowest possible version of the target component in one of a machine level, a process level, or a sub-process level (e.g., P. 4, Sec. of “Versioning and Sharing”, 4th Par. – in .NET, sharing code between applications is an explicit decision; assemblies that are shared have some additional requirements; specifically, shared assemblies should support side by side so multiple versions of the same assembly can be installed and run

Art Unit: 2192

on the same machine, or even within the same process, at the same time; P. 8, Sec. of "Version Policy Levels" – including "Application-specific Policy", Publisher Policy", and "Machine-wide Policy"; P. 9, Sec. of "Machine-wide Policy" – the final policy level is machine-wide policy; machine-wide policy is stored in *machine.config* which is located in the "config" subdirectory under the .NET Framework install directory).

31. **As to claim 18** (Currently Amended) (incorporating the rejection in claim 1), Eisenbach and Gunderloy do not explicitly disclose the method further comprising identifying a servicing value associated with the requested target component.

However, in an analogous art of *Simplifying Deployment and Solving DLL Hell with the .NET Framework™*, Pratschner discloses the method wherein the requested target component is a library component, the method further comprising identifying a servicing value associated with the requested target component (e.g., P. 8, Sec. of "Custom Version Policy", the vendor of a shared assembly may have shipped a service release to an existing assembly and would like all applications to be using the service release instead of the original version; these scenarios and others are supported in the .NET Framework™ through version policies; P. 11, Sec. of "Deployment" – the .NET Framework™ provides extensive code download services are integration with Windows Installer).

Therefore, it would have been obvious to one of ordinary skill in the art, at the time the invention was made to combine the teachings of Pratschner into the Eisenbach-Gunderloy's system to further provide the method wherein the requested target

Art Unit: 2192

component is a library component, the method further comprising identifying a servicing value associated with the requested target component in Eisenbach-Gunderloy system.

The motivation is that it would further enhance the Eisenbach-Gunderloy's system by taking, advancing and/or incorporating Pratschner's system which offers significant advantages that the CLR records version information between pieces of an application and uses that information at run time to ensure that the proper version of a dependency is loaded; version policies can be used by both application developers and administrators to provide some flexibility in choosing which version of a given assembly is loaded as once suggested by Pratschner (e.g., P. 12, Sec. of "Summary").

32. **As to claim 19** (Currently Amended) (incorporating the rejection in claim 18), Pratschner discloses the method wherein identifying an appropriate version of the target component comprises identifying an updated servicing of a target component (e.g., P. 7, Sec. of "Version Policy" – including "Default Version Policy", "Custom Version Policy", and Version Policy Levels"; P. 8, Sec. of "Custom Version Policy", the vendor of a shared assembly may have shipped a service release to an existing assembly and would like all applications to be using the service release instead of the original version; these scenarios and others are supported in the .NET Framework™ through version policies; P. 11, Sec. of "Deployment" – the .NET Framework™ provides extensive code download services are integration with Windows Installer).

Art Unit: 2192

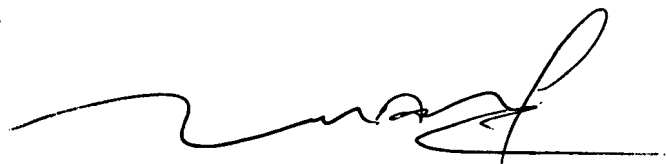
Conclusion

33. Any inquiry concerning this communication or earlier communications from the examiner should be directed to Ben C. Wang whose telephone number is 571-270-1240. The examiner can normally be reached on Monday - Friday, 8:00 a.m. - 5:00 p.m., EST.

If attempts to reach the examiner by telephone are unsuccessful, the examiner's supervisor, Tuan Q. Dam can be reached on 571-272-3695. The fax phone number for the organization where this application or proceeding is assigned is 571-273-8300.

Information regarding the status of an application may be obtained from the Patent Application Information Retrieval (PAIR) system. Status information for published applications may be obtained from either Private PAIR or Public PAIR. Status information for unpublished applications is available through Private PAIR only. For more information about the PAIR system, see <http://pair-direct.uspto.gov>. Should you have questions on access to the Private PAIR system, contact the Electronic Business Center (EBC) at 866-217-9197 (toll-free). If you would like assistance from a USPTO Customer Service Representative or access to the automated information system, call 800-786-9199 (IN USA OR CANADA) or 571-272-1000.

BCW *BW*



TUAN DAM
SUPERVISORY PATENT EXAMINER

November 19, 2007